Decomposition methods for large-scale programming II: some (new) tools

Antonio Frangioni

Dipartimento di Informatica, Università di Pisa

ÖGOR Workshop for PhD students 2022 ‘Krems an der Donau — August 1, 2022
Meta–Outline

- Part I: basic (old) theory
- Part II: some (new) tools
- Part III: advanced (not-so-old) theory
- Part IV: a challenge to you
Outline – Part II

1. Decomposition-aware modelling systems
2. SMS++: design goals
3. SMS++: basic components
4. SMS++: existing Block and Solver
5. A glimpse to some results
6. SMS++: (some of) the missing pieces
7. Conclusions (for now)
1 Decomposition-aware modelling systems

2 SMS++: design goals

3 SMS++: basic components

4 SMS++: existing Block and Solver

5 A glimpse to some results

6 SMS++: (some of) the missing pieces

7 Conclusions (for now)
Modelling languages, and what they are for

- Most interactions with optimization solvers via Algebraic Modelling Languages (AML): commercial AMPL\[^1\], GAMS\[^2\], AIMMS\[^2\] and OPL\[^4\], or open-source Coliop\[^5\] and ZIMPL\[^6\]

- Interfaced with a varying set (few/many) of general-purpose solvers for large problem classes (MILP, MINLP, conic, . . .)

- AML is a separate language, typically interpreted (not efficient)

- Mostly “flat” languages (no OOP), modularity an issue

- Focus on “model once, solve once”; some offer some support for iterative procedures but clearly an afterthought

- Hide the complexities of the model/solution process to inexperienced users

---

Modelling systems, and what they are for

- Modelling systems: libraries written in general-purpose languages providing similar functionalities to AML


- May not fully replicate AML constructs, sometimes more limited

- Solver interfacing and overhead lower with efficient languages (C++)

- Multiple models and iterative procedures more natural

- Can exploit OOP features of host language for better modularity

- Mostly focus on general-purpose solvers and “model once, solve once”

- Tailored for end-users, not algorithms developers

[^7]: https://github.com/coin-or/FlopCpp
[^8]: https://github.com/coin-or/Rehearse
[^9]: https://github.com/coin-or/Gravity
[^10]: https://github.com/coin-or/pulp
[^11]: http://www.pyomo.org
[^12]: https://github.com/jump-dev/JuMP.jl
[^13]: https://yalmip.github.io
Decomposition/structure-aware solvers

- Some solvers provide decomposition capabilities:
  - Cplex does Benders’, structure automatic or user hints
  - SCIP\cite{13} does B&C&P (one-level D-W), pricing & reformulation up to the user (plugins)
  - GCG\cite{13} extends SCIP with automatic and user-defined (one-level) D-W and recently also a generic (one-level) Benders’ approach
  - DDSIP\cite{14} and PIPS\cite{15} implement D-W for two-stage stochastic programs
  - The BaPCoD B&C&P code has been used to develop Coluna.jl\cite{16}, doing one-level D-W and (alpha) Benders’, multi-level planned

- Other solvers use structure in different ways: BlockIP\cite{17}, OOPS\cite{18}

---

[16] https://github.com/atoptima/Coluna.jl
Decomposition-aware modelling systems: are there any?

- **In a word?**

*OOPS* is interfaced with SML\[19\], providing some parallel capabilities. PIPS is interfaced with StructJuMP\[20\], using BlockDecomposition\[21\].

No modelling system is focused on multi-level structure, non-general-purpose solvers, parallel, and modularity/extendability.

Although JuMP is doing a good job at promoting some of these.

We tried working with Julia, but most solvers are in C/C++, and the full circle Julia $\rightarrow$ C++ $\rightarrow$ Julia did not work well.

So we choose no-performance-compromise C++, accepting the drawbacks.

\[19\] https://www.maths.ed.ac.uk/ERGO/sml

\[20\] https://github.com/StructJuMP/StructJuMP.jl

\[21\] https://github.com/atoptima/BlockDecomposition.jl
Decomposition-aware modelling systems: are there any?

- In a word? Very few (that’s two words . . .)
- OOPS is interfaced with SML\[^{19}\], providing some parallel capabilities
- PIPS is interfaced with StructJuMP\[^{20}\], using BlockDecomposition\[^{21}\]
Decomposition-aware modelling systems: are there any?

- In a word? Very few (that's two words . . .)

- OOPS is interfaced with SML\textsuperscript{[19]}, providing some parallel capabilities

- PIPS is interfaced with StructJuMP\textsuperscript{[20]}, using BlockDecomposition\textsuperscript{[21]}

- No modelling system is focused on \textit{multi-level} structure, \textit{non-general-purpose} solvers, \textit{parallel}, and \textit{modularity/extendability}

- Although JuMP is doing a good job at promoting some of these

---

\textsuperscript{[19]} https://www.maths.ed.ac.uk/ERGO/sml

\textsuperscript{[20]} https://github.com/StructJuMP/StructJuMP.jl

\textsuperscript{[21]} https://github.com/atoptima/BlockDecomposition.jl
Decomposition-aware modelling systems: are there any?

- In a word? Very few (that’s two words . . .)

- OOPS is interfaced with SML\textsuperscript{[19]}, providing some parallel capabilities

- PIPS is interfaced with StructJuMP\textsuperscript{[20]}, using BlockDecomposition\textsuperscript{[21]}

- No modelling system is focused on multi-level structure, non-general-purpose solvers, parallel, and modularity/extendability

- Although JuMP is doing a good job at promoting some of these

- We tried working with Julia, but most solvers are in C / C++, and the full circle \texttt{Julia \rightarrow C++ \rightarrow Julia} did not work well

- So we choose no-performance-compromise C++, accepting the drawbacks

\textsuperscript{[19]} \url{https://www.maths.ed.ac.uk/ERGO/sml}
\textsuperscript{[20]} \url{https://github.com/StructJuMP/StructJuMP.jl}
\textsuperscript{[21]} \url{https://github.com/atoptima/BlockDecomposition.jl}
Outline

1. Decomposition-aware modelling systems

2. SMS++: design goals

3. SMS++: basic components

4. SMS++: existing Block and Solver

5. A glimpse to some results

6. SMS++: (some of) the missing pieces

7. Conclusions (for now)
https://gitlab.com/smspp/smspp-project

“For algorithm developers, from algorithm developers”

Open source (LGPL3)

Version 0.5.1 (still a long way to go)
What **SMS++** is

- A core set of C++-17 classes implementing a modelling system that:
  - explicitly supports the notion of **Block** ≡ nested structure
  - separately provides “semantic” information from “syntactic” details, different formulations (set of constraints/variables) of the same problem
  - allows exploiting specialised **Solver** on **Block** with specific structure
  - (potentially) manages any (sensible) dynamic change in the **Block** beyond “just” generation of constraints/variables
  - supports reformulation/restriction/relaxation of **Block**
  - has built-in parallel processing capabilities
  - **should** be able to deal with almost anything (bilevel, PDE, …)

- An hopefully growing set of specialized **Block** and **Solver**

- In perspective an ecosystem fostering collaboration and code sharing
What SMS++ is not

- **An algebraic modelling language**: Block / Solver are C++ code (although it provides some modelling-language-like functionalities)

- **For the faint of heart**: primarily written for algorithmic experts (although users may benefit from having many pre-defined Block)

- **Stable**: only version 0.5.1, lots of further development ahead, significant changes in interfaces not ruled out, actually expected (although current Block / Solver very thoroughly tested)

- **Interfaced with many existing solvers**: Cplex, SCIP, MCFClass, StOp (although the list should hopefully grow, and there are built-in ones)
Outline

1. Decomposition-aware modelling systems
2. SMS++: design goals
3. SMS++: basic components
4. SMS++: existing Block and Solver
5. A glimpse to some results
6. SMS++: (some of) the missing pieces
7. Conclusions (for now)
A Crude Schematic

Objective

Constraint

Variable

Modification

Solver

{ Modification\textsubscript{i} }

Block

{ Solver\textsubscript{i} }

OF

SC\textsubscript{1} SC\textsubscript{2} ...

DC\textsubscript{1} DC\textsubscript{2} ...

SV\textsubscript{1} SV\textsubscript{2} ...

DV\textsubscript{1} DV\textsubscript{2} ...

physical representation

Block\textsubscript{1} Block\textsubscript{2} ...

abstract representation

A. Frangioni (DI — UniPi)
Block

Block = abstract class representing the general concept of “a (part of a) mathematical model with a well-understood identity”

Each :Block a model with specific structure
(e.g., MCFBlock::Block = a Min-Cost Flow problem)

Physical representation of a Block: whatever data structure is required to describe the instance (e.g., G, b, c, u) and methods to change it

Possibly alternative abstract representation(s) of a Block:

- one Objective (but possibly vector-valued)
- any # of groups of (static) Variable
- any # of groups of std::list of (dynamic) Variable
- any # of groups of (static) Constraint
- any # of groups of std::list of (dynamic) Constraint

groups of Variable/Constraint can be single (std::list) or std::vector (...) or boost::multi_array

Any # of sub-Blocks (recursively), possibly of specific type
(e.g., :MMCFBlock::Block may have k :MCFBlock::Block inside)
**Variable**

- Abstract concept, thought to be extended (a matrix, a function, ...)
- Does **not even** have a value
- Knows which **Block** it belongs to
- Can be **fixed** and **unfixed** to/from its current value (whatever that is)
- **Influences** a set of Constraint/Objective/Function
  (actually, a set of :ThinVarDepInterface)
- **Fundamental design decision:** "name" of a Variable = its memory address $\implies$ copying a Variable makes a different Variable $\implies$
dynamic Variables always live in `std::lists`
- **VariableModification:** Modification (fix/unfix)
ThinVarDepInterface

- Generic concept of “something depending on a set of Variable”

- Specific implementation demanded to derived classes for efficiency

- Variable can be added/removed, accessed by index or pointer

- General mechanism for removing Variable (:Modification issued), adding Variable depends on derived class

- “Abstract” STL-like iterator and const-iterator for access

- Other specific methods to describe/search/map the set

- Specific twist: a :ThinVarDepInterface is constructed after and destructed before “its” Variable, clear() method to avoid un-necessary data structure updating during destruction
ThinComputeInterface

- Generic concept of “something that can take time to compute()”
- Specific provisions for the fact that the compute() can:
  - end in several ways (OK, error, stopped, ...) and be resumed
  - be influenced by int/double/std::string parameters which can be gathered in a ComputeConfig:Configuration object (flexible)
  - terminate with approximate solutions, defined in quite general way[22]
  - produce user-defined events triggered by conditions (time, iterations, ...)
- Defaults so that “simple” objects with no parameter do nothing
- Clear rules about effect of changes in the underlying object during and after compute() to allow for “reoptimization”
- Changes may be “explicit” (a Modification issued) or “implicit” (changing a Variable value do not trigger a Modification)
- General asynchronous compute() returning a std::future

[22] van Ackooij, F. “Incremental Bundle Methods Using Upper Models” SIOPT, 2018
Constraint

- Abstract concept, thought to be extended (any algebraic constraint, a matrix constraint, a PDE constraint, bilevel program, ...)
- Depends from a set of Variable (:ThinVarDepInterface)
- Either satisfied or not by the current value of the Variable, checking requires (costly) compute() (:ThinComputeInterface)
- Knows which Block it belongs to
- Can be relaxed and enforced
- Fundamental design decision: “name” of a Constraint = its memory address → copying a Constraint makes a different Constraint → dynamic Constraints always live in std::lists
- ConstraintModification:Modification (relax/enforce)
Objective

- Abstract concept, does not specify its return value (vector, set, …)
- Either minimized or maximized
- Depends from a set of Variable (:ThinVarDepInterface)
- Must be compute()-d w.r.t. the current value of the Variable, possibly a costly operation (:ThinComputeInterface)
- RealObjective: Objective implements “value is an extended real”
- Knows which Block it belongs to
- Same fundamental design decision … (but there is no such thing as a dynamic Objective)
- ObjectiveModification: Modification (change verse)
Real-valued Function

**Depends from** a set of Variable (:ThinVarDepInterface)

**Must be** compute()-d w.r.t. the current value of the Variable, possibly a costly operation (:ThinComputeInterface)

Approximate computation supported (:ThinComputeInterface)

**FunctionModification[Variables] for “easy” changes** $\iff$ reoptimization (shift, adding/removing “quasi separable” Variable)
C05Function and C15Function

- C05Function/C15Function deal with 1st/2nd order information (not necessarily continuous)
- General concept of “diagonal linearization” (gradient, convex/concave/Clarke subgradient, ...) or “vertical linearization” (feasibility cut)
- Multiple linearizations produced at each evaluation (local pool)
- **Global pool of linearizations for reoptimization:**
  - convex combination of linearizations
  - “important linearization” (at optimality)
- C05FunctionModification[Variables/LinearizationShift] for “easy” changes \(\Rightarrow\) reoptimization (linearizations shift, some linearizations entries changing in simple ways)
- C15Function supports (partial) Hessians
- Arbitrary hierarchy of :Function possible/envisioned, any one that makes sense for application and/or solution method
Closer to the ground

ColVariable:Variable: “value = one single real” (possibly $\in \mathbb{Z}$, possibly with $\leq / \geq 0 / \pm 1$ inherent bound constraints)

RowConstraint:Constraint: “$l \leq a \text{ real} \leq u$” $\Rightarrow$ has dual variable (single real) attached to it

OneVarConstraint:RowConstraint: “a real” = a single ColVariable $\equiv$ general bound constraints

FRowConstraint:RowConstraint: “a real” given by a Function

FRealObjective:RealObjective: “value” given by a Function

LinearFunction:Function: a linear form in ColVariable

DQuadFunction:Function: a separable quadratic form

Many things missing (AlgebraicFunction, DenseLinearFunction, Matrix/VectorVariable, …)
Block and Solver

- Any # of Solver attached to a Block to solve it
- Solver for a specific Block can use the physical representation
  \[ \Rightarrow \] no need for explicit Constraint
  \[ \Rightarrow \] abstract representation of Block only constructed on demand

- However, Variable are always present to interface with Solver
  (this may change with physical solution concept, under development)

- A general-purpose Solver uses the abstract representation

- Dynamic Variable/Constraint can be generated on demand
  (user cuts/lazy constraints/column generation)

- For a Solver attached to a Block:
  - Variable not belonging to the Block are constants
  - Constraint not belonging to the Block are ignored

  (belonging = declared there or in any sub-Block recursively)

- Objective of sub-Blocks summed to that of father Block if has same
  verse, otherwise min/max

A. Frangioni (DI — UniPi) Decomposition Methods II ÖGOR PhD School, 2022 22 / 45
Solver

- Solver = interface between a Block and algorithms solving it
- Solver:ThinComputeInterface, inherits and extends interface
- Each Solver attached to a single Block, from which it picks all the data, but any # of Solver can be attached to the same Block
- Solutions are written directly into the Variable of the Block
- Individual Solver can be attached to sub-Block of a Block
- Tries to cater for all the important needs:
  - optimal and sub-optimal solutions, provably unbounded/unfeasible
  - time/resource limits for solutions, but restarts (reoptimization)
  - any # of multiple solutions produced on demand
  - lazily reacts to changes in the data of the Block via Modification
- Slanted towards RealObjective (≈optimality = up/low bounds)
- CDASolver:Solver is “Convex Duality Aware”: bounds are associated to dual solutions (possibly, multiple)
Any change is communicated to each *interested Solver* (attached to the Block or any of its ancestor) via a Modification object

\[ \text{anyone\_there()} \equiv \exists \text{ interested Solver (Modification needed)} \]

Too coarse mechanism, must be improved ("types" of Modification)

Currently, (only) two different kinds of Modification (what changes):

- physical Modification, only specialized Solver concerned
- abstract Modification, only Solver using it concerned

Abstract Modification used to keep both representations in sync

\[ \Rightarrow \text{ a single change may trigger more than one Modification} \]

\[ \Rightarrow \text{ concerns Block() mechanism to avoid this to repeat} \]

\[ \Rightarrow \text{ parameter in changing methods to avoid useless Modification} \]

Specialized Solver disregard abstract Modification and vice-versa

A Block declares which abstract changes it supports

(throws std::exception if the wrong things change)
Modification

- Almost empty base class, then everything has its own derived ones (e.g., added/deleted dynamic Variable/Constraint)

- In general tells **what changes** but **not how** (go read the Block)

- Each Solver has the **responsibility** of cleaning up its list of Modification (smart pointers $\rightarrow$ memory eventually released)

- Solver supposedly **reoptimize** to improve efficiency, which is easier if you can see all list of changes at once (lazy update)

- GroupModification to (recursively) pack many Modification together $\implies$ different “channels” in Block

- Consistency easily ensured if Modification **processed in the arrival order**

- A Solver may optimize the changes (Modifications may cancel each outer out ...), but **its responsibility**
Change (new)

- New concept just introduced in 5.2.0
- Change ≈ Modification + data: tells both what changes and how
- Can be de/serialize-d, not smart pointer
- Change need be defined for each :Block, typically just calling chg_*(()) methods in specialised interface
- Very nifty concept: undo-Change can be produced when Change apply()-ed to Block
- Not all Change can be undo-Change-d (may be too complex)
- Can be used for very general algorithmic schemes (B&X, ...)
- AbstractChange possible, still to be implemented
Support to (coarse-grained) Parallel Computation

- Block can be (r/w) lock()-ed and read_lock()-ed
- lock()-ing a Block automatically lock()s all inner Blocks
- lock() (but not read_lock()) sets an owner and records its std::thread::id; other lock() from the same thread fail (std::mutex would not work there)
- Similar mechanism for read_lock(), any # of concurrent reads
- Write starvation not handled yet
- A Solver can be “lent an ID” (solving an inner Block)
- The list of Modification of Solver is under an “active guard” (std::atomic)
- Distributed computation under development, can exploit general de/serialize Block / Change capabilities
Solution

- Block produces Solution object, possibly using its sub-Blocks’
- Solution can read() its own Block and write() itself back
- Solution is Block-specific rather than Solver-specific
- Solution may save dual information
- Solution may save only a specific subset of primal/dual information
- Linear combination of Solution supported $\implies$ “less general”
- Solution may (automatically) convert in “more general” ones
- Like Block, Solution are tree-structured complex objects

ColVariableSolution: Solution uses the abstract representation of any Block that only have (std::vector or boost::multi_array of) (std::list of) ColVariables to read/write the solution

RowConstraintSolution: Solution same for dual information (RowConstraint), ColRowSolution for both
Configuration

- Block a **tree-structured complex object** →
  - **Configuration** for them a (possibly) tree-structured complex object

- But also **SimpleConfiguration<T>**:
  - `T` an int, a double, a `std::pair<>`, ...

- **[C/O/R]BlockConfiguration**: set [recursively]:
  - which dynamic Variable/Constraint are generated, how (Solver, time limit, parameters ...)
  - which Solution is produced (what is saved)
  - the **ComputeConfiguration**: of any Constraint/Objective that needs one
  - a bunch of other Block parameters

- **[R]BlockSolverConfiguration**: set [recursively] which Solver are attached to the Block and their **ComputeConfiguration**

- Can be **clear()-ed** for cleanup
Often reformulation crucial, but also relaxation or restriction: `get_R3_Block()` produces one, possibly using sub-Blocks’

Obvious special case: copy (clone) should always work

Available R³ Blocks: Block-specific, a :Configuration needed

R³ Block completely independent (new Variable/Constraint), useful for algorithmic purposes (branch, fix, solve, …)

Solution of R³ Block useful to Solver for original Block: `map_back_solution()` (best effort in case of dynamic Variable)

Sometimes keeping R³ Block in sync with original necessary: `map_forward_Modification()`, task of original Block

`map_forward_solution()` and `map_back_Modification()` useful, e.g., dynamic generation of Variable/Constraint in the R³ Block

:Block is in charge of all this, thus decides what it supports
A lot of other support stuff

- All tree-structured complex objects (Block, Configuration, ...) and Solver have an (almost) automatic factory

- All tree-structured complex objects (...) have methods to serialize/deserialize themselves to netCDF files

- A methods factory for changing the physical representation without knowing of which :Block it exactly is (standardised interface)

- AbstractBlock for constructing a model a-la algebraic language, can be derived for “general Block + specific part”

- PolyhedralFunction[Block], very useful for decomposition

- AbstractPath for indexing any Constraint/Variable in a Block

- FakeSolver:Solver stashes away all Modification, UpdateSolver:Solver immediately forwards/R^3Bs them

- ...
Outline

1. Decomposition-aware modelling systems
2. SMS++: design goals
3. SMS++: basic components
4. SMS++: existing Block and Solver
5. A glimpse to some results
6. SMS++: (some of) the missing pieces
7. Conclusions (for now)
Main Existing Block

- MCFBlock/MMCFBlock: single/multicommodity flow
- BinaryKnapsackBlock (actually mixed-integer)
- CapacitatedFacilityLocationBlock (didactic)
- UCBlock for UC, several UnitBlock and NetworkBlock for components
- LagBFunction: \{C05Function, Block\} transforms any Block (with appropriate Objective) into its dual function
- BendersBFunction: \{C05Function, Block\} transforms any Block (with appropriate Constraint) into its value function
- StochasticBlock implements realizations of scenarios into any Block (using methods factory)
- SDDPBlock represents multi-stage stochastic programs suitable for Stochastic Dual Dynamic Programming
Main “Basic” Solver

- **MCFSolver**: templated wrapper to MCFClass\(^{[23]}\) for MCFBlock
- **ThermalUnitDPSolver** for ThermalUnitBlock (state-of-the-art)
- **MILPSolver**: constructs matrix-based representation of any “LP” Block: ColVariable, FRowConstraint, FRealObjective with LinearFunction or DQuadFunction
- **CPXMILPSolver**: MILPSolver and SCIPMILPSolver: MILPSolver wrappers for Cplex and SCIP (to be improved)
- **BundleSolver**: CDASolver: SMS++-native version of\(^{[24]}\) (still shares some code, dependency to be removed), optimizes any (sum of) C05Function, several (but not all) state-of-the-art tricks
- **SDDPSolver**: wrapper for SDDP solver St0pt\(^{[25]}\) using StochasticBlock, BendersBFunction and PolyhedralFunction
- **SDDPGreedySolver**: greedy forward simulator for SDDPBlock

---

\(^{[23]}\) [https://github.com/frangio68/Min-Cost-Flow-Class](https://github.com/frangio68/Min-Cost-Flow-Class)  
\(^{[24]}\) [https://gitlab.com/frangio68/ndosolver_fioracle_project](https://gitlab.com/frangio68/ndosolver_fioracle_project)  
\(^{[25]}\) [https://gitlab.com/stochastic-control/St0pt](https://gitlab.com/stochastic-control/St0pt)
Our Masterpiece: LagrangianDualSolver

- Works for any Block with natural block-diagonal structure: no Objective or Variable, all Constraint linking the inner Block.

- Using LagBFunction stealthily constructs the Lagrangian Dual w.r.t. linking Constraint, $R^3B$-ing or “stealing” the inner Block.

- Solves the Lagrangian Dual with appropriate CDASolver (e.g., but not necessarily, BundleSolver), provides dual and “convexified” solution in original Block.

- Can attach LagrangianDualSolver and (say) :MILPSolver to same Block, solve in parallel!

- Weeks of work in days/hours (if Block of the right form already)

- Hopefully soon BendersDecompositionSolver (crucial component BendersBFunction existing and tested)

- Multilevel nested parallel heterogeneous decomposition by design (but I’ll believe it when I’ll see it running)
Outline

1. Decomposition-aware modelling systems
2. SMS++: design goals
3. SMS++: basic components
4. SMS++: existing Block and Solver
5. A glimpse to some results
6. SMS++: (some of) the missing pieces
7. Conclusions (for now)
Unit Commitment

- Schedule a **set of generating units** to satisfy the **demand** at each node of the **transmission network** at each **time instant** of the horizon.

- Several types of **almost independent blocks** + linking constraints.

- Perfect structure for **Lagrangian relaxation**.

- UCBlock + ThermalUnitBlock, HydroUnitBlock, ... + DCNetworkBlock, EnergyCommunityNetworkBlock ...
Unit Commitment – some results

- **ThermalUnitBlock** provides different formulations, one “exact”

- **ThermalUnitDPSolver** provides efficient solution of 1-UC problems

- **LagrangianDualSolver** (using BundleSolver) + **ThermalUnitDPSolver** provides best trade-off between bound tightness and computational cost as size of the instance grows

<table>
<thead>
<tr>
<th>units</th>
<th>3bin</th>
<th>DP</th>
<th>pt</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>gap</td>
<td>time</td>
<td>gap</td>
</tr>
<tr>
<td>10</td>
<td>0.21</td>
<td>1.03</td>
<td>78.56</td>
<td>0.67</td>
</tr>
<tr>
<td>20</td>
<td>0.90</td>
<td>0.93</td>
<td>480.02</td>
<td>0.51</td>
</tr>
<tr>
<td>50</td>
<td>4.18</td>
<td>0.81</td>
<td>3836.78</td>
<td>0.08</td>
</tr>
</tbody>
</table>

- Just changing few lines in the **BlockConfig** and **BlockSolverConfig**
Seasonal Storage Valuation

- Mid-term (1y) cost-optimal management of water levels in reservoirs considering uncertainties (inflows, temperatures, demands, ...)

- Very large size, nested structure

- Perfect structure for Stochastic Dual Dynamic Programming

- SDDPBlock with as many sub-Block as periods, a StochasticBlock inside each LagBFunction, dynamic PolyhedralFunction to represent the (approximate) value-of-water function, one UCBBlock inside each one
Seasonal Storage Valuation – some results

- SDDPSolver requires convex problem: any of the above

- Brazilian hydro-heavy system: 53 hydro (3 cascade), 98 thermal (coal, gas, nuclear), stochastic inflows (20 scenarios)

- Out-of-sample simulation: 1000 scenarios

<table>
<thead>
<tr>
<th></th>
<th>Cont. relax.</th>
<th>Lag. relax.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost: Avg. / Std.</td>
<td>4.6023e+9 / 1.3608e+9</td>
<td>4.5860e+9 / 1.3556e+9</td>
</tr>
</tbody>
</table>

- Just changing a few lines in the configuration (Lagrangian about 4 times slower, but can be improved)
Investment Layer

- Long-term (30y) optimal (cost, pollution, CO$_2$ emissions, ...) planning of production/transmission investments considering multi-level uncertainties scenarios (technology, economy, politics, ...)

- Many scenarios, huge size, multiple nested structure $\Rightarrow$ multiple nested Benders’ or Lagrangian decomposition and/or SDDP

- Ad-hoc concept for “multiple copies of a block”, could be generalised

- Rather nasty with SMS++, at all doable without?
Investment Layer – Some Results

- **Simplified version**: solve SDDP only once, run optimization with fixed value-of-water function + simulation (GreedySDDPolver)

- EdF EU scenario: 11 nodes (France, Germany, Italy, Switzerland, Eastern Europe, Benelux, Iberia, Britain, Balkans, Baltics, Scandinavia)

- units: 1183 battery, 7 hydro, 518 thermal, 40 intermittent + 20 lines

- 78 weeks hourly (168h), 37 scenarios (demand, inflow, RES generation)

- Investments: 3 thermal units + 2 transmission lines.

- Average cost: original 6.5103e+12, optimized 5.6429e+12

- Running time: 10 hours (4 scenarios in parallel + ParallelBundleSolver with 6 threads)

- Booked some 15000 CPU/h on HPC system for “real” runs
Outline

1. Decomposition-aware modelling systems
2. SMS++: design goals
3. SMS++: basic components
4. SMS++: existing Block and Solver
5. A glimpse to some results
6. SMS++: (some of) the missing pieces
7. Conclusions (for now)
The many things that we do not have (yet)

- A relaxation-agnostic Branch-and-X Solver (Federica to the rescue!)
- Many other forms of (among many other things):
  - Variable (Vector/MatrixVariable, FunctionVariable, …)
  - Constraint (SOCConstraint, SDPConstraint, PDEConstraint, BilevelConstraint, EquilibriumConstraint, …)
  - Objective (RealVectorObjective, …)
  - Function (AlgebraicFunction, …)
- Better handling of many things (groups of stuff, Modification, …)
- Interfaces with many other general-purpose solvers (GuRoBi, OSISolverInterface, Couenne, OR-tools CP-SAT Solver, …)
- Many many many more: Block and their specialised Solver
- Translation layers from real modelling languages (AMPL, JuMP, …)
- In a word: users/mindshare – chicken-and-egg problem
Conclusions

(Part II)
Conclusions and (a lot of) future work

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- **Could** become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
Conclusions and (a lot of) future work

- **SMS++** is there, actively developed
- Perhaps already useful for some fringe use cases
- **Could** become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- **Hefty**, very likely rather unrealistic, sought-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♥ coding, breaks my ♥)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
  - a step towards a reformulation-aware modelling system[26]

---


A. Frangioni (DI — UniPi) Decomposition Methods II ÖGOR PhD School, 2022 45 / 45
Conclusions and (a lot of) future work

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- **Could** become really useful *after having attracted mindshare*, self-reinforcing loop (very hard to start)
- **Hefty, very likely rather unrealistic, sought-after impacts:**
  - improve collaboration and code reuse, reduce huge code waste
    (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
  - a step towards a reformulation-aware modelling system\[26\]
- Not the only game in town, but aiming for a big slice of the cake
Conclusions and (a lot of) future work

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- **Could** become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- **Hefty**, very likely rather unrealistic, sought-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
  - a step towards a reformulation-aware modelling system\[26\]
- Not the only game in town, but aiming for a big slice of the which (hopefully) is **not a lie 😊😊😊**
Conclusions and (a lot of) future work

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- **Could** become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- **Hefty**, very likely rather unrealistic, sought-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I 💖 coding, breaks my 💖)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
  - a step towards a reformulation-aware modelling system[^26]

- Not the only game in town, but aiming for a big slice of the which (hopefully) is not a lie 😊😊😊

- Lots of fun to be had, all contributions welcome